



DHB 1/3/02 3382-47280 90795 MS80683.1

AF/2700

PATENT  
Attorney Reference Number 3382-47280

IN THE UNITED STATES PATENT AND TRADEMARK OFFICE

In re application of: Limprecht et al.

Art Unit: 2151

Application No. 08/959,149

CERTIFICATE OF MAILING

Filed: October 28, 1997

For: SERVER APPLICATION COMPONENTS  
WITH CONTROL OVER STATE  
DURATION

I hereby certify that this paper and the documents referred to as being attached or enclosed herewith are being deposited with the United States Postal Service on January 3, 2002, as First Class Mail in an envelope addressed to: BOX AF, ASSISTANT COMMISSIONER FOR PATENTS, WASHINGTON D.C. 20231.

Examiner: Lao, Sue

  
Stephen A. Wight, Attorney for Applicant

Date: January 3, 2002

TRANSMITTAL LETTER

BOX AF  
ASSISTANT COMMISSIONER FOR PATENTS  
WASHINGTON, DC 20231


RECEIVED  
JAN 29 2002  
Technology Center 2100

Enclosed for filing in the application referenced above are the following:

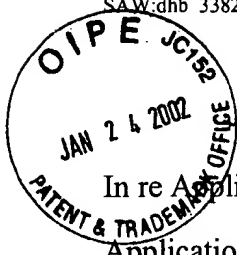
- ☒ Appeal Brief (in triplicate).
- ☒ A check in the amount of \$320.00 to cover filing an Appeal Brief under 37 C.F.R. 1.17(c).
- ☒ The Director is hereby authorized to charge any additional fees that may be required, or credit over-payment, to Account No. 02-4550. A copy of this sheet is enclosed.
- ☒ Please return the enclosed postcard to confirm that the items listed above have been received.

Respectfully submitted,

KLARQUIST SPARKMAN, LLP

By   
Stephen A. Wight  
Registration No. 37,759

One World Trade Center, Suite 1600  
121 S.W. Salmon Street  
Portland, Oregon 97204  
Telephone: (503) 226-7391  
Facsimile: (503) 228-9446  
cc: Docketing  
Client



IN THE UNITED STATES PATENT AND TRADEMARK OFFICE

In re Application of: Limprecht et al.

Art Unit: 2151

Application No.: 08/959,149

Filed: October 28, 1997

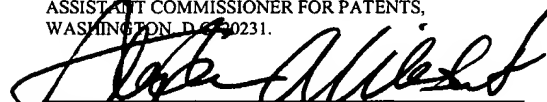
For: SERVER APPLICATION COMPONENTS  
WITH CONTROL OVER STATE DURATION

Examiner: LAO, S

Date: January 3, 2002

CERTIFICATE OF MAILING

I hereby certify that this paper and the documents referred to as being attached or enclosed herewith are being deposited with the United States Postal Service on January 3, 2002 as First Class Mail in an envelope addressed to: BOX AF, ASSISTANT COMMISSIONER FOR PATENTS, WASHINGTON, DC 20231.

  
Stephen A. Wright, Attorney for Applicant

APPEAL BRIEF

RECEIVED

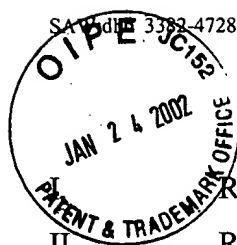
JAN 29 2002

Technology Center 2100

BOX AF  
ASSISTANT COMMISSIONER FOR PATENTS  
Washington, DC 20231

Sir:

This brief is in furtherance of the Notice of Appeal filed November 5, 2001. The fee required under 37 CFR 1.17(b) is enclosed.



## CONTENTS

I.	REAL PARTY IN INTEREST	1
II.	RELATED APPEALS AND INTERFERENCES	1
III.	STATUS OF CLAIMS	1
IV.	STATUS OF AMENDMENTS	1
V.	SUMMARY OF THE INVENTION	1
VI.	ISSUES	2
VII.	GROUPING OF CLAIMS	2
VIII.	ARGUMENT	3
A.	Rejection of Claims under 35 U.S.C. § 103	3
	Claim 1	3
	Claim 5	10
	Claim 13	17
	Claim 18	24
	Claim 21	31
		38
B.	Rejection of Claims under 35 U.S.C. § 103	38
	Claim 2	38
IX.	CONCLUSION	40
	APPENDIX A:	41

**RECEIVED**  
**JAN 29 2002**  
**Technology Center 2100**

## **I. REAL PARTY IN INTEREST**

The real party in interest is Microsoft Corporation, by an assignment from the inventors recorded at Reel 9017, Frame 0250.

## **II. RELATED APPEALS AND INTERFERENCES**

An appeal was filed on October 19, 2001, in co-pending application No. 08/959,139, filed October 28, 1997, having at least some inventors in common.

## **III. STATUS OF CLAIMS**

Claims 1-7, 11-14, and 18-21 are finally rejected and appealed. Claims 8-10 and 15-17 are objected to. Claims 22-24 are allowed.

## **IV. STATUS OF AMENDMENTS**

An amendment after final was filed on September 4, 2001, but not entered.

## **V. SUMMARY OF THE INVENTION**

As opposed to conventional object-oriented programming models where state duration is controlled by the client releasing its reference to the server application component instance, the illustrated embodiment allows an application component to maintain some control over its state duration. *See* Specification, page 5, lines 13-15. An operating service provides an interface that allows the application component to request destruction of its state. *Id.* at page 5, lines 27-28; and at page 6, lines 3-4.

The component's state thus is not left consuming resources on return from the client's call, while awaiting its final release by the client program. *Id.* at page 6, lines 16-18. This allows the server application component to have a lifetime that is independent of the client program's reference to the component. *Id.* at page 23, lines 3-5.

The component is said to be "activated" when the component's state is in existence, and "deactivated" when the component's state is destroyed. *Id.* at page 23, 28-29. While deactivated, the client program retains its reference to the server application component. *Id.* at page 24, lines 26-29. Thus, application component's state is destroyed, potentially while the client maintains a

reference to the component and without action by the client in releasing the application component, which is antithetical to the object-oriented cannon. *Id.* at page 5, lines 13-17. In the illustrated embodiment, the application component may request deactivation before returning from processing the client program's call. *Id.* at page 33, lines 30-32. In response to the request, a server executive 80 deactivates the component, causing its state to be destroyed. *Id.* at page 37, lines 7-9.

In one embodiment, an application component is executing under control of an operating service, the application component having a state and function code for performing work responsive to method invocations from a client. The operating service provides an interface to receive an indication from the application component. In absence of the indication from the application component at the provided interface, the state of the application component is maintained in memory. However, once an indication is received from the application component at the provided interface, the state of the application component is destroyed without action by the client.

## **VI. ISSUES**

A. Whether claims 1-7, 11-14, 18-21 are patentable under 35 U.S.C. §103(a) over the Object Management Group, "The Common Object Request Broker: Architecture and Specification CORBA," Revision 2.0, July 1995 ("CORBA") in view of United States Patent No. 5,889,957, to Ratner et al. ("Ratner") and Steinman, J., "Incremental State Saving in Speedes Using C++," Proceedings of the 1993 Winter Simulation Conference ("Steinman").

B. Whether claim 2 is patentable under 35 U.S.C. §103(a) over CORBA, Ratner, and Steinman in view of United States Patent No. 7,765,174, to Bishop et al. ("Bishop").

## **VII. GROUPING OF CLAIMS**

For reasons detailed below, each of the independent claims 1, 5, 13, 18 and 21 is separately patentable. The dependent claims each contain limitations that further distinguish over the art of record. However, since the limitations of the independent claims are sufficient to distinguish this art and to facilitate the Board's consideration of this appeal, Applicants group the claims for purposes of this appeal as follows.

The patentability of claims 3-4 stand or fall with the patentability of claim 1.

Claim 2 stands or falls alone.

The patentability of claims 6, 7, 11 and 12 stand or fall with the patentability of claim 5.

The patentability of claim 14 stands or falls with the patentability of claim 13.

The patentability of claims 19-20 stand or fall with the patentability of claim 18.

Claim 21 stands or falls alone.

## **VIII. ARGUMENT**

### **A. Rejection of Claims Under 35 U.S.C. 103**

#### **Claim 1**

Applicants respectfully request reversal of the Examiner's rejection of claim 1, because the Examiner failed to establish a *prima facie* case of obviousness. To establish a *prima facie* case of obviousness, three basic criteria must be met. First, there must be some suggestion or motivation, either in the references themselves or in the knowledge generally available to one of ordinary skill in the art, to modify the reference or to combine reference teachings. Second, there must be a reasonable expectation of success. Finally, the prior art reference (or references when combined) must teach or suggest all the claim limitations. (MPEP § 2142.) The art cited by the Examiner fails to support the Examiner's rejection of claim 1.

Claim 1 is generally directed to a method of enhancing scalability of server applications and recites,

1. (Thrice Amended) In a computer having a main memory, a method of enhancing scalability of server applications, comprising:

executing an application component under control of an operating service, the application component having a state and function code for performing work responsive to method invocations from a client;

providing an interface for the operating service to receive an indication from the application component that the work is complete;

maintaining the state in the main memory between the method invocations of the function code by the client in the absence of the indication from the application component that the work is complete; and

destroying the state of the application component in response to the indication from the application component to the operating service at the provided interface that the work is complete and without action by the client. (Emphasis added.)

For example, the Application with emphasis added states,

...As opposed to conventional object-oriented programming models where state duration is controlled solely by the client releasing its reference to the server application component instance.... Page 5, line 13-15.

...the framework provides for server application component control over state duration ... Page 5, line 27-28.

...the component calls framework-provided interfaces... Page 6, 3-4.

...The component's state thus is not left consuming resources on return from the client's call, while awaiting its final release by the client program.... Page 6, line 16-18.

...allows the server application component to have a lifetime that is independent of the client program's reference to the component... Page 23, lines 3-5.

...the component is said to be "activated" when the component's state is in existence, and "deactivated" when the component's state is destroyed.... Page 23, 28-29.

...While deactivated, the client program 134 retains its reference to the server application component 86 indirectly through the transaction server executive 80 (i.e., the safe reference described above). ... Page 24, lines 26-29.

...the server application component 86 may request deactivation before returning from processing the client program's call... Page 33, lines 30-32.

...On return from the client's method call, the transaction server executive 80 deactivates the component, causing its state to be destroyed.... Page 37, lines 7-9.

Thus, the server application component can request destruction of its own state at an interface provided by the operating service and without any action by the client.

The Examiner asserts that the claimed arrangement is obvious in light of a CORBA-Steinman-Ratner combination. Applicants disagree. A CORBA-Ratner-Steinman combination does not teach or suggest destroying the state of the application component in response to the indication from the application component to the operating service at the provided interface ... without action by the client.

First, the Examiner admits that CORBA fails to teach or suggest the recited arrangement. *See* Office Action, mailed July 3, 2001, page 2, ¶ 3. Second, the Examiner does not allege that Steinman teaches or suggests the recited arrangement. Since the references when combined “must teach or suggest all the claim limitations,” Ratner must teach or suggest the recited arrangement or the Examiner’s proposed combination fails. But since Ratner also fails to teach or suggest “destroying the state of the application component in response to the indication from the application component to the operating service at the provided interface ... without action by the client,” the combination fails.

For example, the Examiner asserts that the following Ratner passages, teach or suggest the recited arrangement. *See* Office Action, mailed July 3, 2001, page 3, ¶ 5.

If at any time the server replies with an error message other than the special error code (ERROR 70 or FeContinue), which is the code for “continue this session” (continue), a reply by the server is made to the client, but the connection is immediately thereafter broken between server and client. Thereafter, when the connection is broken, the next time the client establishes a connection it may well be with a different server. (Col. 4, lines 50-55).

...

the receiving process should reply with either FeOK or FeContinue return codes.

A return code of FeOK will cause the run-time environment to break the connection between the originating process and the receiving process, and the originating process, upon detecting the FeOK return code, will call SERVERCLASS\_DIALOG\_END\_ to clean up resources in the originating process's address space. (Col. 7, lines 28-35).

...

However, if and when an FeOK error code message is issued by the server and received by the client (via Linkmon), not only are linkages (or bindings) 603 and 605 broken, but also linkages 607 are broken, and furthermore the Dialog Control Block is deallocated and the LACB is marked available. This effectively kills the context sensitive dialogue. (Col. 10, lines 16-22).

...

sending, by the server, an abort message that ends the dialog. (Col. 12, lines 2-3).



Ratner discusses the creation and destruction of a connection called a context sensitive dialogue. The dialogue insures that all messages flow between an originating process and a receiving process for the duration of the dialogue. *See* Ratner, col. 2, lines 5-10. Note however, that the dialogue is neither the originating process nor the receiving process. Rather, a dialogue is merely a logical connection between the client and server process. *See* Ratner, Abstract; *see also* col. 4, lines 34-37. The dialogue supports communication between the processes, it does not contain the state of the processes. The connection may contain information used to direct messages between processes, but the connection (dialog) discussed in Ratner does not contain the “state of the application component,” and the dialog is not calling for the destruction of its own state. *See* Ratner, col. 10, lines 16-29. A skilled artisan reading about an originating or receiving process killing a logical connection could not reasonably be expected to learn “destroying the state of the application component in response to the indication from the application component ... without action by the client.”

Ratner fails to address the limitations recited in claim 1. Claim 1 recites (1) destroying the state of the application component (2) in response to (3) the indication from the application component (4) to the operating service at the provided interface ... (5) without action by the client. Ratner does not teach or suggest this. The Examiner continues to ignore the fact that in Ratner, (1) the server process is not requesting its own destruction, and (2) the dialogue is not requesting its own destruction.

In fact, Ratner merely supports the object-oriented cannon which establishes that resources are requested, utilized and destroyed at the behest of the objects being served. It is antithetical to conventional object-oriented design for (1) a server object to request its own destruction, (2) for a server object to request its own destruction while clients hold a reference, or (3) for server objects to request their own destruction while they have a positive reference count. Rather, an object is garbage collected by a run-time system after a client has released a reference to the object. For example, Bishop discloses that objects should not be destroyed without the permission of a client holding a reference to the object. *See* Bishop, col. 4, lines 16-22; col. 1, lines 33-35 and lines 49-55. Thus, the prior art relied upon by the Office actually teaches away

from “destroying the state of the application component in response to the indication from the application component.”

Ratner reinforces the object-oriented cannon. Applicants are unable to find any statement in Ratner that teaches or suggests an application component calling an interface to destroy its own state and “without action by the client.” Ratner discloses that an “originating process” can create a dialogue, and can abort a dialogue. *See* col. 7, line 1; col. 7, line 43. Further, a “receiving process” can break a dialogue with a return code of FeOK. *See* col. 7, line 30. However, nowhere does Ratner teach or suggest that a dialogue can request destruction of its own state. Nowhere does Ratner suggest that the server process can request its own state’s destruction. The Office has failed to cite any reference that teaches or suggests “destroying the state of the application component in response to the indication from the application component ... without action by the client.”

In the discussion of claim 21, the Office asserts that Ratner discloses that the “server triggers the breaking of dialog and destruction/deallocation of server state (information maintained in the Dialogue Control Block; and a session/dialog is broken immediately after the server sends out a message other than the special error code (which is the code for “continue this session”).” *See* Office Action, page 3, lines 1-4.

Thus, the Examiner’s position equates a “Dialogue Control Block” with “the state of the application component.” Applicants disagree. As understood by Applicants, the Dialog Control Block (DCB) discussed in Ratner, is used to identify and direct context sensitive messages from a client to a server. *See* Ratner, col. 9, lines 39-57. However, in Ratner, the DCB is not disclosed as maintaining a server application component’s state. Ratner does not teach that the state of the application component is destroyed in response to an indication from the application component at the provided interface.

Ratner, at most, is disclosing that an originating or receiving process can cause destruction of a DCB. However, nowhere does Ratner teach or suggest “destroying” an application component in response to its own indication. In Ratner, the DCB is a data structure that is destroyed based on a request of a process. Destruction of a DCB data structure (which merely supports a context sensitive path send) based on the request of a process, fails to teach or

suggest “destroying the state of the application component in response to the indication from the application component ... without action by the client.”

Further, Ratner describes termination of a communications session by a server, as follows:

... the ability of the runtime to bind the originating process to the receiving process for the duration of the dialogue ....

Column 2, lines 6-7.

...The originating process issues the SERVERCLASS\_DIALOG\_BEGIN\_ procedure call, targeting a specific type of receiving process. The run-time environment will begin a receiving process and create a logical connection between the two cooperating processes (the originating process and the receiving process)....

Column 7, lines 14-20.

...A dialogue (session) can be broken by either a server or a client....

Column 4, lines 36-37.

... a sort of temporary dedicated communication between client and server is established....

Column 9, lines 52-54.

... the first part of the protocol is the response by a server to a Dialogue Begin or Send. Most of the time the server response is “FeContinue” (which causes a session to be maintained)....

Column 9, lines 58-61.

...a return code of FeContinue will cause the run-time environment to maintain the connection between the originating process and the receiving process ...

Column 10, lines 1-4.

... However, if and when an FeOK error code message is issued by the server and received by the client (via Linkmon), not only are linkages (or bindings) 603 and 605 broken, but also linkages 607 are broken, and furthermore the Dialog Control Block is deallocated and the LACB is marked available. This effectively kills the context sensitive dialogue....

Column 10, lines 17-23.

Thereafter, when the connection is broken, the next time the client establishes a connection it may well be with a different server.

Column 4, lines 53-55.

As understood by Applicant, Ratner fails to teach or suggest “destroying the state of the application component in response to the indication from the application component to the operating service at the provided interface that the work is complete and without action by the

client.” The recited passage in Ratner indicates that the “message is issued by the server and received by the client.” As understood by Applicant, the client (originating process) ends a “dialog” by calling “SERVERCLASS\_DIALOG\_END\_” (col. 7, lines 1-8), and the server ends a dialogue by responding to a client message with a return message (“FeOK”) that is received by the client. Col. 10, lines 17-23. Further, the server can respond to other client actions with a return message which continues the session (“FeContinue”). In either case, the client takes action. Accordingly, Ratner fails to teach “destroying the state of the application component ... in response to the indication from the application component,” and “without action by the client.”

The Examiner overlooks the fact that Ratner teaches away from no client action. *See* Office Action, mailed July 7, 2001, page 7, lines 2-5. It is true that a return code of FeOK will trigger the break down of a dialogue, but the return code is being “returned to” the client process, which in response thereto calls “SERVERCLASS\_DIALOG\_END\_.” Col. 7, lines 32-35 (“the originating process upon detecting the FeOK return code, will call SERVERCLASS\_DIALOG\_END\_ to clean up resources in the originating process’s address space.”) Thus Ratner requires action by the client to clean up dialogue resources existing on the client side.

The Examiner fails to consider the fact that Ratner discloses the clean up of dialogue resources on both the client and server side. Again, the FeOK return code is being returned to the client process, and the client process requests the breakdown of client resources. Col. 7, lines 32-35. It is clear in Ratner that “client action” is required to receive the FeOK return code, and to call the dialogue end procedure. Thus, Ratner discloses client action, and teaches away from the recited arrangement.

The Examiner states that the interface recited in claim 1 is disclosed in Ratner. *See* Office Action, mailed July 7, 2001, page 6, ¶ 3 (“the provided interface as claimed is met by the API of Ratner”). This is not so. The interface described in Ratner is not “an interface” called by an application component indicating the destruction of itself. Claim 1 recites “providing an interface for the operating service to receive an indication from the application component that the work is complete ... [and] destroying the state of the application component in response to

the indication from the application component to the operating service at the provided interface and without action by the client.”

Ratner discusses an interface (col. 7, lines 4-7) called by an originating process or client (col. 7, lines 14-15, 32-35, 37-41, 43-44; col. 8, lines 10-13), and replied to by a receiving process using return codes and read functions to communicate with the originating process. Col. 7, lines 28-29; col. 8, lines 25-27. Thus the Ratner interface is used by the client and server to create and destroy a context sensitive path send dialogue. Since Ratner does not describe “an interface” called by an application component indicating the destruction of itself, this element has not been shown by the Examiner.

Since neither CORBA, Steinman, nor Ratner teaches or suggests “destroying the state of the application component in response to the indication from the application component to the operating service at the provided interface that the work is complete and without action by the client,” a CORBA-Steinman-Ratner combination also fails to teach or suggest the recited arrangement.

For at least these reasons, claim 1 is separately patentable over this art, and the Examiner’s rejection of claim 1 should be reversed.

#### Claim 5

Applicants respectfully request reversal of the Examiner’s rejection of claim 5, because the Examiner failed to establish a *prima facie* case of obviousness. To establish a *prima facie* case of obviousness, three basic criteria must be met. First, there must be some suggestion or motivation, either in the references themselves or in the knowledge generally available to one of ordinary skill in the art, to modify the reference or to combine reference teachings. Second, there must be a reasonable expectation of success. Finally, the prior art reference (or references when combined) must teach or suggest all the claim limitations. (MPEP § 2142.) The art cited by the Examiner fails to support the Examiner’s rejection of claim 5.

Claim 5 is generally directed to a computer operating environment for scalable, component-based server applications and recites,

5. (Unchanged) In a computer, a computer operating environment for scalable, component-based server applications, comprising:

a run-time service for executing an application component in a process, the application component having a state and implementing a set of functions;

an instance creation service operative, responsive to a request of a client, to return a reference to the application component through the run-time service to the client, whereby the client calls functions of the application component indirectly through the run-time service using the reference to initiate work by the application component; and

the run-time service being operative, responsive to an indication from the application component that the application component has completed the work for the client, to destroy the application component's state on the application component returning from a call by the client without action by the client. (Emphasis added.)

For example, the Application with emphasis added states,

...As opposed to conventional object-oriented programming models where state duration is controlled solely by the client releasing its reference to the server application component instance....

Page 5, line 13-15.

...the framework provides for server application component control over state duration ...

Page 5, line 27-28.

...the component calls framework-provided interfaces...

Page 6, 3-4.

...The component's state thus is not left consuming resources on return from the client's call, while awaiting its final release by the client program....

Page 6, line 16-18.

...allows the server application component to have a lifetime that is independent of the client program's reference to the component...

Page 23, lines 3-5.

...the component is said to be "activated" when the component's state is in existence, and "deactivated" when the component's state is destroyed....

Page 23, 28-29.

...While deactivated, the client program 134 retains its reference to the server application component 86 indirectly through the transaction server executive 80 (i.e., the safe reference described above). ...

Page 24, lines 26-29.

...the server application component 86 may request deactivation before returning from processing the client program's call...

Page 33, lines 30-32.

...On return from the client's method call, the transaction server executive 80 deactivates the component, causing its state to be destroyed.... Page 37, lines 7-9.

Thus, the server application component can request the run-time service to destroy the application component's state and without action by the client.

The Examiner asserts that the claimed arrangement is obvious in light of a CORBA-Steinman-Ratner combination. Applicants disagree. A CORBA-Ratner-Steinman combination does not teach or suggest "the run-time service being operative, responsive to an indication from the application component that the application component has completed the work for the client, to destroy the application component's state on the application component returning from a call by the client without action by the client."

First, the Examiner admits that CORBA fails to teach or suggest the recited arrangement. *See* Office Action, mailed July 3, 2001, page 2, ¶ 3. Second, the Examiner does not allege that Steinman teaches or suggests the recited arrangement. Since the references when combined "must teach or suggest all the claim limitations," Ratner must teach or suggest the recited arrangement or the Examiner's proposed combination fails. But since Ratner also fails to teach or suggest "the run-time service being operative, responsive to an indication from the application component that the application component has completed the work for the client, to destroy the application component's state on the application component returning from a call by the client without action by the client," the combination fails.

For example, the Examiner asserts that the following Ratner passages, teach or suggest the recited arrangement. *See* Office Action, mailed July 3, 2001, page 3, ¶ 5.

If at any time the server replies with an error message other than the special error code (ERROR 70 or FeContinue), which is the code for "continue this session" (continue), a reply by the server is made to the client, but the connection is immediately thereafter broken between server and client. Thereafter, when the connection is broken, the next time the client establishes a connection it may well be with a different server. (Col. 4, lines 50-55).

...  
the receiving process should reply with either FeOK or FeContinue return codes.

A return code of FeOK will cause the run-time environment to break the connection between the originating process and the receiving process, and the originating process, upon detecting the FeOK return code, will call `SERVERCLASS_DIALOG_END` to clean up resources in the originating process's address space. (Col. 7, lines 28-35).

...

However, if and when an FeOK error code message is issued by the server and received by the client (via Linkmon), not only are linkages (or bindings) 603 and 605 broken, but also linkages 607 are broken, and furthermore the Dialog Control Block is deallocated and the LACB is marked available. This effectively kills the context sensitive dialogue. (Col. 10, lines 16-22).

...

sending, by the server, an abort message that ends the dialog. (Col. 12, lines 2-3).

Ratner discusses the creation and destruction of a connection called a context sensitive dialogue. The dialogue insures that all messages flow between an originating process and a receiving process for the duration of the dialogue. *See* Ratner, col. 2, lines 5-10. Note however, that the dialogue is neither the originating process nor the receiving process. Rather, a dialogue is merely a logical connection between the client and server process. *See* Ratner, Abstract; *see also* col. 4, lines 34-37. The dialogue supports communication between the processes, it does not contain the state of the processes. The connection may contain information used to direct messages between processes, but the connection (dialog) discussed in Ratner does not contain the "state of the application component," and the dialog is not calling for the destruction of its own state. *See* Ratner, col. 10, lines 16-29. A skilled artisan reading about an originating or receiving process killing a logical connection could not reasonably be expected to learn "the run-time service being operative, responsive to an indication from the application component that the application component has completed the work for the client, to destroy the application component's state on the application component returning from a call by the client without action by the client."

Ratner fails to address the limitations recited in claim 5. Claim 5 recites (1) the run-time being operative to destroy the application component's state (2) responsive to an indication from the application component (3) that the application component has completed the work (4)



without action by the client. Ratner does not teach or suggest this. The Examiner continues to ignore the fact that in Ratner, (1) the application component is not requesting its own destruction, and (2) the dialogue is not requesting its own destruction.

In fact, Ratner merely supports the object-oriented cannon which establishes that resources are requested, utilized and destroyed at the behest of the objects being served. It is antithetical to conventional object-oriented design for (1) a server object to request its own destruction, (2) for a server object to request its own destruction while clients hold a reference, or (3) for server objects to request their own destruction while they have a positive reference count. Rather, an object is garbage collected by a run-time system after a client has released a reference to the object. For example, Bishop discloses that objects should not be destroyed without the permission of a client holding a reference to the object. *See* Bishop, col. 4, lines 16-22; col. 1, lines 33-35 and lines 49-55. Thus, the prior art relied upon by the Office actually teaches away from “the run-time service being operative, responsive to an indication from the application component that the application component has completed the work for the client, to destroy the application component's state on the application component returning from a call by the client without action by the client.”

Ratner reinforces the object-oriented cannon. Applicants are unable to find any statement in Ratner that teaches or suggests an application component indicating its own destruction and “without action by the client.” Ratner discloses that an “originating process” can create a dialogue, and can abort a dialogue. *See* col. 7, line 1; col. 7, line 43. Further, a “receiving process” can break a dialogue with a return code of FeOK. *See* col. 7, line 30. However, nowhere does Ratner teach or suggest that a dialogue can request destruction of its own state. Nowhere does Ratner suggest that the server process can request its own state's destruction. The Office has failed to cite any reference that teaches or suggests “the run-time service being operative, responsive to an indication from the application component that the application component has completed the work for the client, to destroy the application component's state on the application component returning from a call by the client without action by the client.”

In the discussion of claim 21, the Office asserts that Ratner discloses that the “server triggers the breaking of dialog and destruction/deallocation of server state (information maintained in the Dialogue Control Block; and a session/dialog is broken immediately after the

server sends out a message other than the special error code (which is the code for “continue this session”).” *See* Office Action, page 3, lines 1-4.

Thus, the Examiner’s position equates a “Dialogue Control Block” with “the state of the application component.” Applicants disagree. As understood by Applicants, the Dialog Control Block (DCB) discussed in Ratner, is used to identify and direct context sensitive messages from a client to a server. *See* Ratner, col. 9, lines 39-57. However, in Ratner, the DCB is not disclosed as maintaining a server application component’s state. Ratner does not teach that the state of the application component is destroyed in response to an indication from the application component.

Ratner, at most, is disclosing that an originating or receiving process can cause destruction of a DCB. However, nowhere does Ratner teach or suggest “destroying” an application component in response to its own indication. In Ratner, the DCB is a data structure that is destroyed based on a request of a process. Destruction of a DCB data structure (which merely supports a context sensitive path send) based on the request of a process, fails to teach or suggest “the run-time service being operative, responsive to an indication from the application component that the application component has completed the work for the client, to destroy the application component's state on the application component returning from a call by the client, without action by the client.”

Further, Ratner describes termination of a communications session by a server, as follows:

... the ability of the runtime to bind the originating process to the receiving process for the duration of the dialogue ....

Column 2, lines 6-7.

...The originating process issues the SERVERCLASS\_DIALOG\_BEGIN\_ procedure call, targeting a specific type of receiving process. The run-time environment will begin a receiving process and create a logical connection between the two cooperating processes (the originating process and the receiving process)....

Column 7, lines 14-20.

...A dialogue (session) can be broken by either a server or a client....

Column 4, lines 36-37.

... a sort of temporary dedicated communication between client and server is established....

Column 9, lines 52-54.

... the first part of the protocol is the response by a server to a Dialogue Begin or Send. Most of the time the server response is "FeContinue" (which causes a session to be maintained)....

Column 9, lines 58-61.

...a return code of FeContinue will cause the run-time environment to maintain the connection between the originating process and the receiving process ...

Column 10, lines 1-4.

... However, if and when an FeOK error code message is issued by the server and received by the client (via Linkmon), not only are linkages (or bindings) 603 and 605 broken, but also linkages 607 are broken, and furthermore the Dialog Control Block is deallocated and the LACB is marked available. This effectively kills the context sensitive dialogue....

Column 10, lines 17-23.

...

Thereafter, when the connection is broken, the next time the client establishes a connection it may well be with a different server.

Column 4, lines 53-55.

As understood by Applicant, Ratner fails to teach or suggest "the run-time service being operative, responsive to an indication from the application component that the application component has completed the work for the client, to destroy the application component's state on the application component returning from a call by the client without action by the client." The recited passage in Ratner indicates that the "message is issued by the server and received by the client." As understood by Applicant, the client (originating process) ends a "dialog" by calling "SERVERCLASS\_DIALOG\_END\_" (col. 7, lines 1-8), and the server ends a dialogue by responding to a client message with a return message ("FeOK") that is received by the client. Col. 10, lines 17-23. Further, the server can respond to other client actions with a return message which continues the session ("FeContinue"). In either case, the client takes action. Accordingly, Ratner fails to teach "the run-time service being operative, responsive to an indication from the application component that the application component has completed the work for the client, to destroy the application component's state on the application component returning from a call by the client without action by the client."

The Examiner overlooks the fact that Ratner teaches away from no client action. *See* Office Action, mailed July 7, 2001, page 7, lines 2-5. It is true that a return code of FeOK will trigger the break down of a dialogue, but the return code is being "returned to" the client process,

which in response thereto calls "SERVERCLASS\_DIALOG\_END\_." Col. 7, lines 32-35 ("the originating process upon detecting the FeOK return code, will call SERVERCLASS\_DIALOG\_END\_ to clean up resources in the originating process's address space.") Thus Ratner requires action by the client to clean up dialogue resources existing on the client side.

The Examiner fails to consider the fact that Ratner discloses the clean up of dialogue resources on both the client and server side. Again, the FeOK return code is being returned to the client process, and the client process requests the breakdown of client resources. Col. 7, lines 32-35. It is clear in Ratner that "client action" is required to receive the FeOK return code, and to call the dialogue end procedure. Thus, Ratner discloses client action, and teaches away from the recited arrangement.

Since neither CORBA, Steinman, nor Ratner teaches or suggests "the run-time service being operative, responsive to an indication from the application component that the application component has completed the work for the client, to destroy the application component's state on the application component returning from a call by the client without action by the client," a CORBA-Steinman-Ratner combination also fails to teach or suggest the recited arrangement.

For at least these reasons, claim 5 is separately patentable over this art, and the Examiner's rejection of claim 5 should be reversed.

### Claim 13

Applicants respectfully request reversal of the Examiner's rejection of claim 13, because the Examiner failed to establish a *prima facie* case of obviousness. To establish a *prima facie* case of obviousness, three basic criteria must be met. First, there must be some suggestion or motivation, either in the references themselves or in the knowledge generally available to one of ordinary skill in the art, to modify the reference or to combine reference teachings. Second, there must be a reasonable expectation of success. Finally, the prior art reference (or references when combined) must teach or suggest all the claim limitations. (MPEP § 2142.) The art cited by the Examiner fails to support the Examiner's rejection of claim 13.

Claim 13 is generally directed to a method of enhancing scalability of server applications and recites,

13. (Once Amended) In a computer, a method of encapsulating state of processing work for a client by a server application in a component with improved scalability, comprising:

encapsulating function code and a processing state for the work in a component;

providing a reference through an operating service for a client to call the function code of the component to initiate processing of the work by the component;

receiving an indication from the component that the work by the component is complete; and

discarding the processing state of the component responsive to the component indicating completion of the work before receiving any indication from the client that the component's work is complete. (Emphasis added.)

For example, the Application with emphasis added states,

...As opposed to conventional object-oriented programming models where state duration is controlled solely by the client releasing its reference to the server application component instance.... Page 5, line 13-15.

...the framework provides for server application component control over state duration ... Page 5, line 27-28.

...the component calls framework-provided interfaces... Page 6, 3-4.

...The component's state thus is not left consuming resources on return from the client's call, while awaiting its final release by the client program.... Page 6, line 16-18.

...allows the server application component to have a lifetime that is independent of the client program's reference to the component... Page 23, lines 3-5.

...the component is said to be "activated" when the component's state is in existence, and "deactivated" when the component's state is destroyed.... Page 23, 28-29.

...While deactivated, the client program 134 retains its reference to the server application component 86 indirectly through the transaction server executive 80 (i.e., the safe reference described above). ... Page 24, lines 26-29.

...the server application component 86 may request deactivation before returning from processing the client program's call...

Page 33, lines 30-32.

...On return from the client's method call, the transaction server executive 80 deactivates the component, causing its state to be destroyed... Page 37, lines 7-9.

Thus, the server application component can request destruction of its own state at an interface provided by the operating service and without any action by the client.

The Examiner asserts that the claimed arrangement is obvious in light of a CORBA-Steinman-Ratner combination. Applicants disagree. A CORBA-Ratner-Steinman combination does not teach or suggest discarding the processing state of the component responsive to the component indicating completion of the work before receiving any indication from the client that the component's work is complete.

First, the Examiner admits that CORBA fails to teach or suggest the recited arrangement. *See* Office Action, mailed July 3, 2001, page 2, ¶ 3. Second, the Examiner does not allege that Steinman teaches or suggests the recited arrangement. Since the references when combined "must teach or suggest all the claim limitations," Ratner must teach or suggest the recited arrangement or the Examiner's proposed combination fails. But since Ratner also fails to teach or suggest "discarding the processing state of the component responsive to the component indicating completion of the work before receiving any indication from the client that the component's work is complete."

For example, the Examiner asserts that the following Ratner passages, teach or suggest the recited arrangement. *See* Office Action, mailed July 3, 2001, page 3, ¶ 5.

If at any time the server replies with an error message other than the special error code (ERROR 70 or FeContinue), which is the code for "continue this session" (continue), a reply by the server is made to the client, but the connection is immediately thereafter broken between server and client. Thereafter, when the connection is broken, the next time the client establishes a connection it may well be with a different server. (Col. 4, lines 50-55).

...

the receiving process should reply with either FeOK or FeContinue return codes.

A return code of FeOK will cause the run-time environment to break the connection between the originating process and the receiving process, and the originating process,

upon detecting the FeOK return code, will call `SERVERCLASS_DIALOG_END` to clean up resources in the originating process's address space. (Col. 7, lines 28-35).

...

However, if and when an FeOK error code message is issued by the server and received by the client (via Linkmon), not only are linkages (or bindings) 603 and 605 broken, but also linkages 607 are broken, and furthermore the Dialog Control Block is deallocated and the LACB is marked available. This effectively kills the context sensitive dialogue. (Col. 10, lines 16-22).

...

sending, by the server, an abort message that ends the dialog. (Col. 12, lines 2-3).

Ratner discusses the creation and destruction of a connection called a context sensitive dialogue. The dialogue insures that all messages flow between an originating process and a receiving process for the duration of the dialogue. *See* Ratner, col. 2, lines 5-10. Note however, that the dialogue is neither the originating process nor the receiving process. Rather, a dialogue is merely a logical connection between the client and server process. *See* Ratner, Abstract; *see also* col. 4, lines 34-37. The dialogue supports communication between the processes, it does not contain the state of the processes. The connection may contain information used to direct messages between processes, but the connection (dialog) discussed in Ratner does not contain the "state of the application component," and the dialog is not calling for the destruction of its own state. *See* Ratner, col. 10, lines 16-29. A skilled artisan reading about an originating or receiving process killing a logical connection could not reasonably be expected to learn "discarding the processing state of the component responsive to the component indicating completion of the work before receiving any indication from the client that the component's work is complete."

Ratner fails to address the limitations recited in claim 13. Claim 13 recites (1) discarding the processing state of the component (2) responsive to (3) the component indicating completion of the work (4) before receiving any indication from the client that the component's work is complete. Ratner does not teach or suggest this. The Examiner continues to ignore the fact that in Ratner, (1) a component is not requesting its own destruction, and (2) a dialogue is not requesting its own destruction.

In fact, Ratner merely supports the object-oriented cannon which establishes that resources are requested, utilized and destroyed at the behest of the objects being served. It is antithetical to conventional object-oriented design for (1) a server object to request its own destruction, (2) for a server object to request its own destruction while clients hold a reference, or (3) for server objects to request their own destruction while they have a positive reference count. Rather, an object is garbage collected by a run-time system after a client has released a reference to the object. For example, Bishop discloses that objects should not be destroyed without the permission of a client holding a reference to the object. *See* Bishop, col. 4, lines 16-22; col. 1, lines 33-35 and lines 49-55. Thus, the prior art relied upon by the Office actually teaches away from “discarding the processing state of the component responsive to the component indicating completion of the work before receiving any indication from the client that the component’s work is complete.”

Ratner reinforces the object-oriented cannon. Applicants are unable to find any statement in Ratner that teaches or suggests discarding the state of a component responsive to the component’s indication before receiving any indication from the client that work is complete. Ratner discloses that an “originating process” can create a dialogue, and can abort a dialogue. *See* col. 7, line 1; col. 7, line 43. Further, a “receiving process” can break a dialogue with a return code of FeOK. *See* col. 7, line 30. However, nowhere does Ratner teach or suggest that a dialogue can request destruction of its own state. Nowhere does Ratner suggest that the server process can request its own state’s destruction. The Office has failed to cite any reference that teaches or suggests “discarding the processing state of the component responsive to the component indicating completion of the work before receiving any indication from the client that the component’s work is complete.”

In the discussion of claim 21, the Office asserts that Ratner discloses that the “server triggers the breaking of dialog and destruction/deallocation of server state (information maintained in the Dialogue Control Block; and a session/dialog is broken immediately after the server sends out a message other than the special error code (which is the code for “continue this session”).” *See* Office Action, page 3, lines 1-4.

Thus, the Examiner’s position equates a “Dialogue Control Block” with “the state of the application component.” Applicants disagree. As understood by Applicants, the Dialog Control



Block (DCB) discussed in Ratner, is used to identify and direct context sensitive messages from a client to a server. *See* Ratner, col. 9, lines 39-57. However, in Ratner, the DCB is not disclosed as maintaining a server application component's state. Ratner does not teach that the state of the application component is destroyed in response to an indication from the application component at the provided interface.

Ratner, at most, is disclosing that an originating or receiving process can cause destruction of a DCB. However, nowhere does Ratner teach or suggest "discarding the processing state of the component responsive to" its own indication. In Ratner, the DCB is a data structure that is destroyed based on a request of a process. Destruction of a DCB data structure (which merely supports a context sensitive path send) based on the request of a process, fails to teach or suggest "discarding the processing state of the component responsive to the component indicating completion of the work before receiving any indication from the client that the component's work is complete."

Further, Ratner describes termination of a communications session by a server, as follows:

... the ability of the runtime to bind the originating process to the receiving process for the duration of the dialogue ....

Column 2, lines 6-7.

...The originating process issues the SERVERCLASS\_DIALOG\_BEGIN\_ procedure call, targeting a specific type of receiving process. The run-time environment will begin a receiving process and create a logical connection between the two cooperating processes (the originating process and the receiving process)....

Column 7, lines 14-20.

...A dialogue (session) can be broken by either a server or a client....

Column 4, lines 36-37.

... a sort of temporary dedicated communication between client and server is established....

Column 9, lines 52-54.

... the first part of the protocol is the response by a server to a Dialogue Begin or Send. Most of the time the server response is "FeeContinue" (which causes a session to be maintained)....

Column 9, lines 58-61.

...a return code of FeContinue will cause the run-time environment to maintain the connection between the originating process and the receiving process ...

Column 10, lines 1-4.

... However, if and when an FeOK error code message is issued by the server and received by the client (via Linkmon), not only are linkages (or bindings) 603 and 605 broken, but also linkages 607 are broken, and furthermore the Dialog Control Block is deallocated and the LACB is marked available. This effectively kills the context sensitive dialogue....

Column 10, lines 17-23.

...

Thereafter, when the connection is broken, the next time the client establishes a connection it may well be with a different server.

Column 4, lines 53-55.

As understood by Applicant, Ratner fails to teach or suggest “discarding the processing state of the component responsive to the component indicating completion of the work before receiving any indication from the client that the component’s work is complete.” The recited passage in Ratner indicates that the “message is issued by the server and received by the client.” As understood by Applicant, the client (originating process) ends a “dialog” by calling “SERVERCLASS\_DIALOG\_END\_” (col. 7, lines 1-8), and the server ends a dialogue by responding to a client message with a return message (“FeOK”) that is received by the client. Col. 10, lines 17-23. Further, the server can respond to other client actions with a return message which continues the session (“FeContinue”). In either case, the client takes action. Accordingly, Ratner fails to teach “discarding the processing state of the component responsive to the component indicating completion of the work before receiving any indication from the client that the component’s work is complete.”

The Examiner overlooks the fact that Ratner teaches away from no client action. *See* Office Action, mailed July 7, 2001, page 7, lines 2-5. It is true that a return code of FeOK will trigger the break down of a dialogue, but the return code is being “returned to” the client process, which in response thereto calls “SERVERCLASS\_DIALOG\_END\_.” Col. 7, lines 32-35 (“the originating process upon detecting the FeOK return code, will call SERVERCLASS\_DIALOG\_END\_ to clean up resources in the originating process’s address space.”) Thus Ratner requires action by the client to clean up dialogue resources existing on the client side.

The Examiner fails to consider the fact that Ratner discloses the clean up of dialogue resources on both the client and server side. Again, the FeOK return code is being returned to

the client process, and the client process requests the breakdown of client resources. Col. 7, lines 32-35. It is clear in Ratner that “client action” is required to receive the FeOK return code, and to call the dialogue end procedure. Thus, Ratner discloses client action, and teaches away from the recited arrangement.

Ratner discusses an interface (col. 7, lines 4-7) called by an originating process or client (col. 7, lines 14-15, 32-35, 37-41, 43-44; col. 8, lines 10-13), and replied to by a receiving process using return codes and read functions to communicate with the originating process. Col. 7, lines 28-29; col. 8, lines 25-27. Thus the Ratner interface is used by the client and server to create and destroy a context sensitive path send dialogue. Since Ratner does not describe a component indicating the destruction of itself, this element has not been shown by the Examiner.

Since neither CORBA, Steinman, nor Ratner teaches or suggests “discarding the processing state of the component responsive to the component indicating completion of the work before receiving any indication from the client that the component’s work is complete,” a CORBA-Steinman-Ratner combination also fails to teach or suggest the recited arrangement.

For at least these reasons, claim 13 is separately patentable over this art, and the Examiner’s rejection of claim 13 should be reversed.

#### Claim 18

Applicants respectfully request reversal of the Examiner’s rejection of claim 18, because the Examiner failed to establish a *prima facie* case of obviousness. To establish a *prima facie* case of obviousness, three basic criteria must be met. First, there must be some suggestion or motivation, either in the references themselves or in the knowledge generally available to one of ordinary skill in the art, to modify the reference or to combine reference teachings. Second, there must be a reasonable expectation of success. Finally, the prior art reference (or references when combined) must teach or suggest all the claim limitations. (MPEP § 2142.) The art cited by the Examiner fails to support the Examiner’s rejection of claim 18.

Claim 18 is generally directed to a system service for providing an execution environment for scalable application components and recites,

18. (Unchanged) In a computer, a system service for providing an execution environment for scalable application components, comprising:

code responsive to a request from a client program to create an application component for returning to the client program a reference through the system service to the application component;

code responsive to a call from the client program using the reference for initiating processing of work by the application component, the application component producing a processing state during processing the work;

code for receiving an indication from the application component that processing by the application component of the work is complete; and

code for destroying the processing state of the application program without action from the client program. (Emphasis added.)

For example, the Application with emphasis added states,

...As opposed to conventional object-oriented programming models where state duration is controlled solely by the client releasing its reference to the server application component instance.... Page 5, line 13-15.

...the framework provides for server application component control over state duration ... Page 5, line 27-28.

...the component calls framework-provided interfaces... Page 6, 3-4.

...The component's state thus is not left consuming resources on return from the client's call, while awaiting its final release by the client program.... Page 6, line 16-18.

...allows the server application component to have a lifetime that is independent of the client program's reference to the component... Page 23, lines 3-5.

...the component is said to be "activated" when the component's state is in existence, and "deactivated" when the component's state is destroyed.... Page 23, 28-29.

...While deactivated, the client program 134 retains its reference to the server application component 86 indirectly through the transaction server executive 80 (i.e., the safe reference described above). ... Page 24, lines 26-29.

...the server application component 86 may request deactivation before returning from processing the client program's call... Page 33, lines 30-32.

...On return from the client's method call, the transaction server executive 80 deactivates the component, causing its state to be destroyed.... Page 37, lines 7-9.

Thus, the application component can indicate work completion, resulting in destruction of the application component's state and without action from the client program.

The Examiner asserts that the claimed arrangement is obvious in light of a CORBA-Steinman-Ratner combination. Applicants disagree. A CORBA-Ratner-Steinman combination does not teach or suggest "receiving an indication from the application component that processing by the application component of the work is complete; and ... destroying the processing state of the application program without action from the client program."

First, the Examiner admits that CORBA fails to teach or suggest the recited arrangement. *See* Office Action, mailed July 3, 2001, page 2, ¶ 3. Second, the Examiner does not allege that Steinman teaches or suggests the recited arrangement. Since the references when combined "must teach or suggest all the claim limitations," Ratner must teach or suggest the recited arrangement or the Examiner's proposed combination fails. But since Ratner also fails to teach or suggest "receiving an indication from the application component that processing by the application component of the work is complete; and ... destroying the processing state of the application program without action from the client program," the combination fails.

For example, the Examiner asserts that the following Ratner passages, teach or suggest the recited arrangement. *See* Office Action, mailed July 3, 2001, page 3, ¶ 5.

If at any time the server replies with an error message other than the special error code (ERROR 70 or FeContinue), which is the code for "continue this session" (continue), a reply by the server is made to the client, but the connection is immediately thereafter broken between server and client. Thereafter, when the connection is broken, the next time the client establishes a connection it may well be with a different server. (Col. 4, lines 50-55).

...

the receiving process should reply with either FeOK or FeContinue return codes.

A return code of FeOK will cause the run-time environment to break the connection between the originating process and the receiving process, and the originating process, upon detecting the FeOK return code, will call SERVERCLASS\_DIALOG\_END\_ to clean up resources in the originating process's address space. (Col. 7, lines 28-35).

...

However, if and when an FeOK error code message is issued by the server and received by the client (via Linkmon), not only are linkages (or bindings) 603 and 605 broken, but also linkages 607 are broken, and furthermore the Dialog Control Block is deallocated and the LACB is marked available. This effectively kills the context sensitive dialogue. (Col. 10, lines 16-22).

...  
sending, by the server, an abort message that ends the dialog. (Col. 12, lines 2-3).

Ratner discusses the creation and destruction of a connection called a context sensitive dialogue. The dialogue insures that all messages flow between an originating process and a receiving process for the duration of the dialogue. *See* Ratner, col. 2, lines 5-10. Note however, that the dialogue is neither the originating process nor the receiving process. Rather, a dialogue is merely a logical connection between the client and server process. *See* Ratner, Abstract; *see also* col. 4, lines 34-37. The dialogue supports communication between the processes, it does not contain the state of the processes. The connection may contain information used to direct messages between processes, but the connection (dialog) discussed in Ratner does not contain the “state of the application component,” and the dialog is not calling for the destruction of its own state. *See* Ratner, col. 10, lines 16-29. A skilled artisan reading about an originating or receiving process killing a logical connection could not reasonably be expected to learn “receiving an indication from the application component that processing by the application component of the work is complete; and ... destroying the processing state of the application program without action from the client program.”

Ratner fails to address the limitations recited in claim 18. Claim 18 recites (1) receiving an indication from the application component (2) that processing by the application component of the work is complete (3) destroying the processing state of the application program (4) without action from the client program. Ratner does not teach or suggest this. The Examiner continues to ignore the fact that in Ratner, (1) the server process is not requesting its own destruction, and (2) the dialogue is not requesting its own destruction.

In fact, Ratner merely supports the object-oriented cannon which establishes that resources are requested, utilized and destroyed at the behest of the objects being served. It is

antithetical to conventional object-oriented design for (1) a server object to request its own destruction, (2) for a server object to request its own destruction while clients hold a reference, or (3) for server objects to request their own destruction while they have a positive reference count. Rather, an object is garbage collected by a run-time system after a client has released a reference to the object. For example, Bishop discloses that objects should not be destroyed without the permission of a client holding a reference to the object. *See* Bishop, col. 4, lines 16-22; col. 1, lines 33-35 and lines 49-55. Thus, the prior art relied upon by the Office actually teaches away from “receiving an indication from the application component that processing by the application component of the work is complete; and ... destroying the processing state of the application program without action from the client program.”

Ratner reinforces the object-oriented cannon. Applicants are unable to find any statement in Ratner that teaches or suggests receiving an indication from an application component that processing ... is complete and destroying the processing state ... without action from the client program. Ratner discloses that an “originating process” can create a dialogue, and can abort a dialogue. *See* col. 7, line 1; col. 7, line 43. Further, a “receiving process” can break a dialogue with a return code of FeOK. *See* col. 7, line 30. However, nowhere does Ratner teach or suggest that a dialogue can request destruction of its own state. Nowhere does Ratner suggest that the server process can request its own state’s destruction. The Office has failed to cite any reference that teaches or suggests “receiving an indication from the application component that processing by the application component of the work is complete; and ... destroying the processing state of the application program without action from the client program.”

In the discussion of claim 21, the Office asserts that Ratner discloses that the “server triggers the breaking of dialog and destruction/deallocation of server state (information maintained in the Dialogue Control Block; and a session/dialog is broken immediately after the server sends out a message other than the special error code (which is the code for “continue this session”).” *See* Office Action, page 3, lines 1-4.

Thus, the Examiner’s position equates a “Dialogue Control Block” with “the state of the application component.” Applicants disagree. As understood by Applicants, the Dialog Control Block (DCB) discussed in Ratner, is used to identify and direct context sensitive messages from a client to a server. *See* Ratner, col. 9, lines 39-57. However, in Ratner, the DCB is not

disclosed as maintaining a server application component's state. Ratner does not teach that the state of the application component is destroyed in response to an indication from the application component.

Ratner, at most, is disclosing that an originating or receiving process can cause destruction of a DCB. However, nowhere does Ratner teach or suggest "destroying" an application component's state in response to its own indication. In Ratner, the DCB is a data structure that is destroyed based on a request of a process. Destruction of a DCB data structure (which merely supports a context sensitive path send) based on the request of a process, fails to teach or suggest "receiving an indication from the application component that processing by the application component of the work is complete; and ... destroying the processing state of the application program without action from the client program."

Further, Ratner describes termination of a communications session by a server, as follows:

... the ability of the runtime to bind the originating process to the receiving process for the duration of the dialogue ....

Column 2, lines 6-7.

...The originating process issues the SERVERCLASS\_DIALOG\_BEGIN\_ procedure call, targeting a specific type of receiving process. The run-time environment will begin a receiving process and create a logical connection between the two cooperating processes (the originating process and the receiving process)....

Column 7, lines 14-20.

...A dialogue (session) can be broken by either a server or a client....

Column 4, lines 36-37.

... a sort of temporary dedicated communication between client and server is established....

Column 9, lines 52-54.

... the first part of the protocol is the response by a server to a Dialogue Begin or Send. Most of the time the server response is "FeeContinue" (which causes a session to be maintained)....

Column 9, lines 58-61.

...a return code of FeContinue will cause the run-time environment to maintain the connection between the originating process and the receiving process ...

Column 10, lines 1-4.

... However, if and when an FeOK error code message is issued by the server and received by the client (via Linkmon), not only are linkages (or bindings) 603 and 605 broken, but



also linkages 607 are broken, and furthermore the Dialog Control Block is deallocated and the LACB is marked available. This effectively kills the context sensitive dialogue.... Column 10, lines 17-23.

Thereafter, when the connection is broken, the next time the client establishes a connection it may well be with a different server. Column 4, lines 53-55.

As understood by Applicant, Ratner fails to teach or suggest the claimed arrangement. The recited passage in Ratner indicates that the “message is issued by the server and received by the client.” As understood by Applicant, the client (originating process) ends a “dialog” by calling “SERVERCLASS\_DIALOG\_END\_” (col. 7, lines 1-8), and the server ends a dialogue by responding to a client message with a return message (“FeOK”) that is received by the client. Col. 10, lines 17-23. Further, the server can respond to other client actions with a return message which continues the session (“FeContinue”). In either case, the client takes action. Accordingly, Ratner fails to teach “receiving an indication from the application component that processing by the application component of the work is complete; and ... destroying the processing state of the application program *without action from the client program.*”

The Examiner overlooks the fact that Ratner teaches away from no client action. *See* Office Action, mailed July 7, 2001, page 7, lines 2-5. It is true that a return code of FeOK will trigger the break down of a dialogue, but the return code is being “returned to” the client process, which in response thereto calls “SERVERCLASS\_DIALOG\_END\_.” Col. 7, lines 32-35 (“the originating process upon detecting the FeOK return code, will call SERVERCLASS\_DIALOG\_END\_ to clean up resources in the originating process’s address space.”) Thus Ratner requires action by the client to clean up dialogue resources existing on the client side.

The Examiner fails to consider the fact that Ratner discloses the clean up of dialogue resources on both the client and server side. Again, the FeOK return code is being returned to the client process, and the client process requests the breakdown of client resources. Col. 7, lines 32-35. It is clear in Ratner that “client action” is required to receive the FeOK return code, and to call the dialogue end procedure. Thus, Ratner discloses client action, and teaches away from the recited arrangement.

Since neither CORBA, Steinman, nor Ratner teaches or suggests “receiving an indication from the application component that processing by the application component of the work is complete; and ... destroying the processing state of the application program without action from the client program,” a CORBA-Steinman-Ratner combination also fails to teach or suggest the recited arrangement.

For at least these reasons, claim 18 is separately patentable over this art, and the Examiner’s rejection of claim 18 should be reversed.

### Claim 21

Applicants respectfully request reversal of the Examiner’s rejection of claim 21, because the Examiner failed to establish a *prima facie* case of obviousness. To establish a *prima facie* case of obviousness, three basic criteria must be met. First, there must be some suggestion or motivation, either in the references themselves or in the knowledge generally available to one of ordinary skill in the art, to modify the reference or to combine reference teachings. Second, there must be a reasonable expectation of success. Finally, the prior art reference (or references when combined) must teach or suggest all the claim limitations. (MPEP § 2142.) The art cited by the Examiner fails to support the Examiner’s rejection of claim 1.

Claim 21 is generally directed to a method of enhancing scalability of server applications and recites,

21. (Once Amended) In a computer having a main memory, a method of enhancing scalability of server applications, comprising:

executing an application component under control of an operating service, the application component having a state and function code for performing work responsive to method invocations from a client;

maintaining the state in the main memory between the method invocations of the function code by the client in the absence of an indication from the application component that the work is complete; and

destroying the state by the operating service in response to an indication from the application component without action by the client, such that the destroyed state is not persistent.  
(Emphasis added.)

For example, the Application with emphasis added states,

...As opposed to conventional object-oriented programming models where state duration is controlled solely by the client releasing its reference to the server application component instance....

Page 5, line 13-15.

...the framework provides for server application component control over state duration ...

Page 5, line 27-28.

...the component calls framework-provided interfaces...

Page 6, 3-4.

...The component's state thus is not left consuming resources on return from the client's call, while awaiting its final release by the client program....

Page 6, line 16-18.

...allows the server application component to have a lifetime that is independent of the client program's reference to the component...

Page 23, lines 3-5.

...the component is said to be "activated" when the component's state is in existence, and "deactivated" when the component's state is destroyed....

Page 23, 28-29.

...While deactivated, the client program 134 retains its reference to the server application component 86 indirectly through the transaction server executive 80 (i.e., the safe reference described above). ...

Page 24, lines 26-29.

...the server application component 86 may request deactivation before returning from processing the client program's call...

Page 33, lines 30-32.

...On return from the client's method call, the transaction server executive 80 deactivates the component, causing its state to be destroyed....

Page 37, lines 7-9.

The Examiner asserts that the claimed arrangement is obvious in light of a CORBA-Steinman-Ratner combination. Applicants disagree. A CORBA-Ratner-Steinman combination does not teach or suggest destroying the state by the operating service in response to an indication from the application component without action by the client, such that the destroyed state is not persistent.

First, the Examiner admits that CORBA fails to teach or suggest the recited arrangement. *See Office Action*, mailed July 3, 2001, page 2, ¶ 3. Second, the Examiner does not allege that Steinman teaches or suggests the recited arrangement. Since the references when combined "must teach or suggest all the claim limitations," Ratner must teach or suggest the recited arrangement or the Examiner's proposed combination fails. But since Ratner also fails to teach

or suggest destroying the state by the operating service in response to an indication from the application component without action by the client, such that the destroyed state is not persistent,” the combination fails.

For example, the Examiner asserts that the following Ratner passages, teach or suggest the recited arrangement. *See* Office Action, mailed July 3, 2001, page 3, ¶ 5.

If at any time the server replies with an error message other than the special error code (ERROR 70 or FeContinue), which is the code for “continue this session” (continue), a reply by the server is made to the client, but the connection is immediately thereafter broken between server and client. Thereafter, when the connection is broken, the next time the client establishes a connection it may well be with a different server. (Col. 4, lines 50-55).

...

the receiving process should reply with either FeOK or FeContinue return codes.

A return code of FeOK will cause the run-time environment to break the connection between the originating process and the receiving process, and the originating process, upon detecting the FeOK return code, will call SERVERCLASS\_DIALOG\_END\_ to clean up resources in the originating process's address space. (Col. 7, lines 28-35).

...

However, if and when an FeOK error code message is issued by the server and received by the client (via Linkmon), not only are linkages (or bindings) 603 and 605 broken, but also linkages 607 are broken, and furthermore the Dialog Control Block is deallocated and the LACB is marked available. This effectively kills the context sensitive dialogue. (Col. 10, lines 16-22).

...

sending, by the server, an abort message that ends the dialog. (Col. 12, lines 2-3).

Ratner discusses the creation and destruction of a connection called a context sensitive dialogue. The dialogue insures that all messages flow between an originating process and a receiving process for the duration of the dialogue. *See* Ratner, col. 2, lines 5-10. Note however, that the dialogue is neither the originating process nor the receiving process. Rather, a dialogue is merely a logical connection between the client and server process. *See* Ratner, Abstract; *see*

*also* col. 4, lines 34-37. The dialogue supports communication between the processes, it does not contain the state of the processes. The connection may contain information used to direct messages between processes, but the connection (dialog) discussed in Ratner does not contain the “state of the application component,” and the dialog is not calling for the destruction of its own state. *See* Ratner, col. 10, lines 16-29. A skilled artisan reading about an originating or receiving process killing a logical connection could not reasonably be expected to learn “destroying the state by the operating service in response to an indication from the application component without action by the client, such that the destroyed state is not persistent.”

Ratner fails to address the limitations recited in claim 21. Claim 21 recites (1) destroying the state (2) in response to (3) an indication from the application component (4) without action by the client, (5) such that the destroyed state is not persistent. Ratner does not teach or suggest this. The Examiner continues to ignore the fact that in Ratner, (1) the server process is not requesting its own destruction, and (2) the dialogue is not requesting its own destruction.

In fact, Ratner merely supports the object-oriented cannon which establishes that resources are requested, utilized and destroyed at the behest of the objects being served. It is antithetical to conventional object-oriented design for (1) a server object to request its own destruction, (2) for a server object to request its own destruction while clients hold a reference, or (3) for server objects to request their own destruction while they have a positive reference count. Rather, an object is garbage collected by a run-time system after a client has released a reference to the object. For example, Bishop discloses that objects should not be destroyed without the permission of a client holding a reference to the object. *See* Bishop, col. 4, lines 16-22; col. 1, lines 33-35 and lines 49-55. Thus, the prior art relied upon by the Office actually teaches away from “destroying the state by the operating service in response to an indication from the application component without action by the client, such that the destroyed state is not persistent.”

Ratner reinforces the object-oriented cannon. Applicants are unable to find any statement in Ratner that teaches or suggests destroying the state ... in response to an indication from the application component without action by the client. Ratner discloses that an “originating process” can create a dialogue, and can abort a dialogue. *See* col. 7, line 1; col. 7, line 43. Further, a “receiving process” can break a dialogue with a return code of FeOK. *See* col. 7, line

30. However, nowhere does Ratner teach or suggest that a dialogue can request destruction of its own state. Nowhere does Ratner suggest that the server process can request its own state's destruction. The Office has failed to cite any reference that teaches or suggests "destroying the state by the operating service in response to an indication from the application component without action by the client, such that the destroyed state is not persistent."

In the discussion of claim 21, the Office asserts that Ratner discloses that the "server triggers the breaking of dialog and destruction/deallocation of server state (information maintained in the Dialogue Control Block; and a session/dialog is broken immediately after the server sends out a message other than the special error code (which is the code for "continue this session")." *See Office Action*, page 3, lines 1-4.

Thus, the Examiner's position equates a "Dialogue Control Block" with "the state of the application component." Applicants disagree. As understood by Applicants, the Dialog Control Block (DCB) discussed in Ratner, is used to identify and direct context sensitive messages from a client to a server. *See Ratner*, col. 9, lines 39-57. However, in Ratner, the DCB is not disclosed as maintaining a server application component's state. Ratner does not teach that the state of the application component is destroyed in response to an indication from the application component.

Ratner, at most, is disclosing that an originating or receiving process can cause destruction of a DCB. However, nowhere does Ratner teach or suggest "destroying" an application component's state in response to its own indication. In Ratner, the DCB is a data structure that is destroyed based on a request of a process. Destruction of a DCB data structure (which merely supports a context sensitive path send) based on the request of a process, fails to teach or suggest "destroying the state by the operating service in response to an indication from the application component without action by the client, such that the destroyed state is not persistent."

Further, Ratner describes termination of a communications session by a server, as follows:

... the ability of the runtime to bind the originating process to  
the receiving process for the duration of the dialogue ....  
Column 2, lines 6-7.

...The originating process issues the SERVERCLASS\_DIALOG\_BEGIN\_ procedure call, targeting a specific type of receiving process. The run-time environment will begin a receiving process and create a logical connection between the two cooperating processes (the originating process and the receiving process).... Column 7, lines 14-20.

...A dialogue (session) can be broken by either a server or a client.... Column 4, lines 36-37.

... a sort of temporary dedicated communication between client and server is established.... Column 9, lines 52-54.

... the first part of the protocol is the response by a server to a Dialogue Begin or Send. Most of the time the server response is "FeContinue" (which causes a session to be maintained).... Column 9, lines 58-61.

...a return code of FeContinue will cause the run-time environment to maintain the connection between the originating process and the receiving process ... Column 10, lines 1-4.

... However, if and when an FeOK error code message is issued by the server and received by the client (via Linkmon), not only are linkages (or bindings) 603 and 605 broken, but also linkages 607 are broken, and furthermore the Dialog Control Block is deallocated and the LACB is marked available. This effectively kills the context sensitive dialogue.... Column 10, lines 17-23.

...Thereafter, when the connection is broken, the next time the client establishes a connection it may well be with a different server. Column 4, lines 53-55.

As understood by Applicant, Ratner fails to teach or suggest "destroying the state by the operating service in response to an indication from the application component without action by the client, such that the destroyed state is not persistent." The recited passage in Ratner indicates that the "message is issued by the server and received by the client." As understood by Applicant, the client (originating process) ends a "dialog" by calling "SERVERCLASS\_DIALOG\_END\_" (col. 7, lines 1-8), and the server ends a dialogue by responding to a client message with a return message ("FeOK") that is received by the client. Col. 10, lines 17-23. Further, the server can respond to other client actions with a return message which continues the session ("FeContinue"). In either case, the client takes action. Accordingly, Ratner fails to teach "destroying the state by the operating service in response to an indication

from the application component without action by the client, such that the destroyed state is not persistent.”

The Examiner overlooks the fact that Ratner teaches away from no client action. *See* Office Action, mailed July 7, 2001, page 7, lines 2-5. It is true that a return code of FeOK will trigger the break down of a dialogue, but the return code is being “returned to” the client process, which in response thereto calls “SERVERCLASS\_DIALOG\_END\_.” Col. 7, lines 32-35 (“the originating process upon detecting the FeOK return code, will call SERVERCLASS\_DIALOG\_END\_ to clean up resources in the originating process’s address space.”) Thus Ratner requires action by the client to clean up dialogue resources existing on the client side.

The Examiner fails to consider the fact that Ratner discloses the clean up of dialogue resources on both the client and server side. Again, the FeOK return code is being returned to the client process, and the client process requests the breakdown of client resources. Col. 7, lines 32-35. It is clear in Ratner that “client action” is required to receive the FeOK return code, and to call the dialogue end procedure. Thus, Ratner discloses client action, and teaches away from the recited arrangement.

Since neither CORBA, Steinman, nor Ratner teaches or suggests “destroying the state by the operating service in response to an indication from the application component without action by the client, such that the destroyed state is not persistent.” a CORBA-Steinman-Ratner combination also fails to teach or suggest the recited arrangement.

For at least these reasons, claim 21 is separately patentable over this art, and the Examiner’s rejection of claim 21 should be reversed.

#### B. Rejection of Claim 2 Under 35 U.S.C. 103

##### Claim 2

Applicants respectfully request reversal of the Examiner’s rejection of claim 2, because the Examiner failed to establish a *prima facie* case of obviousness. To establish a *prima facie* case of obviousness, three basic criteria must be met. First, there must be some suggestion or motivation, either in the references themselves or in the knowledge generally available to one of ordinary skill in the art, to modify the reference or to combine reference teachings. Second,



there must be a reasonable expectation of success. Finally, the prior art reference (or references when combined) must teach or suggest all the claim limitations. (MPEP § 2142.) The art cited by the Examiner fails to support the Examiner's rejection of claim 2.

Claim 2 depends from claim 1, and is generally directed to a method of enhancing scalability of server applications. Claim 2 is patentable for all the reasons stated above for claim 1. However, claim 2 recites the following additional language:

2. (Once Amended) The method of claim 1 wherein the operating service retains an operating service's reference to the application component and the step of destroying the state comprises releasing the operating service's reference to the application component by the operating service while the client retains a client's reference to the application component. (Emphasis added.)

Claim 2 is allowable for all the reasons stated above in claim 1. First, Bishop fails to teach or suggest the recited arrangement. Although the Office Action asserts that Bishop discusses only the last phrase from claim 2, Applicants discuss Bishop more broadly because Bishop helps establish how the prior art teaches away from all claims, and towards the conventional object oriented canon.

In Bishop the client *allows* the operating service to garbage collect an object by calling the make weak function. See Bishop, col. 1, lines 33-35, and lines 49-55. Therefore, the client must take *action*, and the object being destroyed by garbage collection takes *no action*. Thus, Bishop teaches away from "destroying the state of the application component in response to the indication from the application component ... without action by the client." In Bishop, the operating service garbage collects objects without any indication from the object itself, and only after the client calls the make weak function. Nowhere does Bishop teach or suggest that an application component can request destruction of itself "without action by the client" and while a "client retains a ... reference."

Bishop exemplifies how the prior art views object destruction. It is contrary to the object oriented canon for (1) an object to request its own destruction, (2) for an object to request its own destruction while clients hold a reference, and (3) for objects to be destroyed at their own request while they have a positive reference count. Rather, objects are garbage collected by a run-time

system after clients have released or reduced their holding power on an object. For example, Bishop discloses that objects should not be destroyed without the consent of the client (make weak function) and not while another client maintains a strong reference to the object. *See* Bishop, col. 4, lines 16-22; col. 6, lines 41-43. Bishop discusses allowing garbage collection of an object if a client of the object calls a make weak function. *See* Bishop, col. 1, lines 57-61; col. 7, lines 28-34. Thus, the prior art teaches away from “destroying the state of the application component in response to the indication from the application component ... without action by the client ... while the client retains a client’s reference.”


For at least these reasons, claim 2 is separately patentable over this art, and the Examiner’s rejection of claim 2 should be reversed.

**IX. CONCLUSION**

Accordingly, the rejection of claims 1-7, 11-14, and 18-21 should be reversed, and all claims passed to issue.

Respectfully submitted,

KLARQUIST SPARKMAN CAMPBELL  
LEIGH & WHINSTON, LLP

By   
Stephen A. Wight  
Registration No. 37,759

One World Trade Center, Suite 1600  
121 S.W. Salmon Street  
Portland, Oregon 97204  
Telephone: (503) 226-7391  
Facsimile: (503) 228-9446

cc: Patent Group Docketing Dept. (75706.1)  
KS Paralegal

**APPENDIX A**  
**CLAIMS ON APPEAL**

1. (Thrice Amended) In a computer having a main memory, a method of enhancing scalability of server applications, comprising:

executing an application component under control of an operating service, the application component having a state and function code for performing work responsive to method invocations from a client;

providing an interface for the operating service to receive an indication from the application component that the work is complete;

maintaining the state in the main memory between the method invocations of the function code by the client in the absence of the indication from the application component that the work is complete; and

destroying the state of the application component in response to the indication from the application component to the operating service at the provided interface that the work is complete and without action by the client.

2. (Once Amended) The method of claim 1 wherein the operating service retains an operating service's reference to the application component and the step of destroying the state comprises releasing the operating service's reference to the application component by the operating service while the client retains a client's reference to the application component.

3. (Once Amended) The method of claim 1 wherein the step of destroying the state comprises resetting the state of the application component to the application component's initial post-creation state.

4. (Once Amended) The method of claim 1 wherein said destroying the state is performed by the operating service upon a next return of the application component from the client's call following the indication from the application component that the work is complete.

5. (Unchanged) In a computer, a computer operating environment for scalable, component-based server applications, comprising:

a run-time service for executing an application component in a process, the application component having a state and implementing a set of functions;

an instance creation service operative, responsive to a request of a client, to return a reference to the application component through the run-time service to the client, whereby the client calls functions of the application component indirectly through the run-time service using the reference to initiate work by the application component; and

the run-time service being operative, responsive to an indication from the application component that the application component has completed the work for the client, to destroy the application component's state on the application component returning from a call by the client without action by the client.

6. (Unchanged) The computer operating environment of claim 5 wherein the indication is a call from the client to commit or abort a transaction encompassing the work.

7. (Unchanged) The computer operating environment of claim 5 wherein the application component initiates the indication before returning from the call by the client, whereby the application component's state is destroyed immediately on return from the client's call without further action by the client.

11. (Unchanged) The computer operating environment of claim 5 wherein the run-time service holds a reference to an instance of the application component, and destroys the application component's state by releasing the reference to the instance.

12. (Unchanged) The computer operating environment of claim 5 wherein the run-time service destroys the application component's state by resetting the state.

13. (Once Amended) In a computer, a method of encapsulating state of processing work for a client by a server application in a component with improved scalability, comprising:

encapsulating function code and a processing state for the work in a component;  
providing a reference through an operating service for a client to call the function code of the component to initiate processing of the work by the component;  
receiving an indication from the component that the work by the component is complete; and  
discarding the processing state of the component responsive to the component indicating completion of the work before receiving any indication from the client that the component's work is complete.

14. (Once Amended) The method of claim 13 further comprising:  
performing the step of discarding the processing state upon a next return of the component from a call of the client following the indication from the component that the work is complete.

18. (Unchanged) In a computer, a system service for providing an execution environment for scalable application components, comprising:

code responsive to a request from a client program to create an application component for returning to the client program a reference through the system service to the application component;

code responsive to a call from the client program using the reference for initiating processing of work by the application component, the application component producing a processing state during processing the work;

code for receiving an indication from the application component that processing by the application component of the work is complete; and

code for destroying the processing state of the application program without action from the client program.

19. (Unchanged) The system service of claim 18 further comprising:  
code for producing an instance of the application component and retaining a reference to the instance, the instance containing the processing state; and

wherein the code for destroying the processing state comprises code for releasing the reference to the instance without action from the client program to thereby cause the processing state to be destroyed.

20. (Unchanged) The system service of claim 18 wherein the code for destroying the processing state comprises:

code for resetting the processing state to an initial state of the application component.

21. (Once Amended) In a computer having a main memory, a method of enhancing scalability of server applications, comprising:

executing an application component under control of an operating service, the application component having a state and function code for performing work responsive to method invocations from a client;

maintaining the state in the main memory between the method invocations of the function code by the client in the absence of an indication from the application component that the work is complete; and

destroying the state by the operating service in response to an indication from the application component without action by the client, such that the destroyed state is not persistent.

Claims 22-24 are allowed.